UNIVERSITAT POLITECNICA DE
CATALUNYA

PARALLELISM

Roger Vilaseca Darne and Xavier Martin Ballesteros
PAR4110

Lab 2: Brief tutorial on OpenMP programming model

20th March 2019, Q1

Contents

1 Introduction

2 OpenMP questionnaire

2.1 Parallel regions

2.1.1 1.hello.c

1. How many times will you see the "Hello world!" message if the program is executed with "./1.hello"?

If the program is executed with the given code, we see 24 times the message "Hello world!" because the number of threads has been set to 24. We can see it if we add the line **printf("%d", omp_get_num_threads());** in the code.

2. Without changing the program, how to make it to print 4 times the "Hello World!" message?

We can change the number of threads to execute the code with the command **OMP_NUM_THREADS=4 ./1.hello**. This will execute the program 1.hello.c using only 4 threads.

2.1.2 2.hello.c

1. Is the execution of the program correct? (i.e., prints a sequence of "(Thid) Hello (Thid) world!" being Thid the thread identifier). If not, add a data sharing clause to make it correct? TORNAR A REPASSSSSSSSSAR!!!!!!!!!!!!!!!!!!!!!!!! The execution of the program is not correct. This is because there is no flow control inside the parallel region, so the 8 threads execute at the same time the code inside this region. Consequently, there may be some prints that are not ordered. It also may occur that in the same line appears "(Thread X) Hello (Thread Y) Hello..." without the second part of the print.

In order to make it correct, we have to add the **critical** clause as shown below. This clause specifies that code inside its region is executed by one thread at a time.

```
#pragma omp parallel num_threads(8)
#pragma omp critical
{
    id =omp_get_thread_num();
    printf("(%d) Hello ",id);
    printf("(%d) world!\n",id);
}
```

2. Are the lines always printed in the same order? Why the messages sometimes appear intermixed? (Execute several times in order to see this).

The lines are not always printed in the same order. This is because threads do not always arrive at the same moment in the critical region. Hence, in one execution it may happen that thread number 2 is the first to arrive to the critical

part, so it will enter the first (preventing the other threads to enter the region) and printing correctly the first message, and in another execution thread number 2 may be the last one to enter the critical part, being the last to output the "Hello world!" message.

2.1.3 3.how many.c

Assuming the OMP NUM THREADS variable is set to 8 with "export_OMP_NUM_THREADS=8"

1. How many "Hello world ..." lines are printed on the screen?

Using 8 threads, the different "Hello world..." messages appear a total of 20 times.

- Hello world from the first parallel (8)! => 8 times
- Hello world from the second parallel (2)! => 2 times
- Hello world from the second parallel (3)! => 3 times
- Hello world from the third parallel (4)! => 4 times
- Hello world from the fourth parallel (3)! => 3 times

2. What does omp_get_num_threads return when invoked outside and inside a parallel region?

Outside the parallel region the function returns 1 because, as we are in a sequential part, there is only 1 thread executing the code.

There are several parallel regions (3 with #pragma omp parallel and 1 with #pragma omp parallel num_threads(4)). The first region returns 8 threads, because we set the number of threads to use to 8. The second region is inside the first loop of the main. Before calling #pragma omp parallel, we set the number of threads to i, which is the variable used in the for. This for is executed 2 times. In the first one, we set the number of threads to 2, so the value will be 2 threads. In the next loop iteration, the i value is 3, so we will have 3 threads executing the parallel region. In the next parallel region, we set with *num_threads(4)* the number of threads to execute the code, so the value in this region will be 4. Finally, the last parallel region will be executed by 3 threads. Altough we previously set the number of threads to be 8, we changed later this value inside the loop. The last value was 3, so in this new region the value will still be 3.

2.1.4 4.data sharing.c

1. Which is the value of variable x after the execution of each parallel region with different datasharing attributes (shared, private, firstprivate and reduction)? Is that the value you would expect? (Execute several times if necessary)

The first datasharing attribute is **shared(x)**. This attribute makes variable x to be shared by all the threads that are executing the code of this region. As we are modifying its value, it is possible that 2 threads read at the same time the x value and later write in it their computed value. In this situation, the final value will not be the correct, as we have not added some values. This is what occurs sometimes. The value of this part was 107, similar as what we expected.

The second one is **private(x)**. This attribute specifies that each thread should have its own instance of the variable x, initialised with a random value. Once we leave the parallel region, the x value is again the one before executing this region. The value is 5, which is the expected value, because the print is done outside the parallel region and the value of x was 5.

The next attribute, **firstprivate(x)**, is very similar to private. The main difference is that in this attribute the variable is initialized with the value of the variable, because it exists before the parallel construct. We print the value outside the parallel region, so the value is again 5.

The last attribute is **reduction(+:x)**. It specifies that variable x that is private to each thread is the subject of a reduction operation at the end of the parallel region. It is initialized to the neutral value of the operation (sum in this case). The value is what we expected, 125 because the variable is private for each thread.

2.2 Loop parallelism

2.2.1 1.schedule.c

1. Which iterations of the loops are executed by each thread for each schedule kind?

In **schedule(static)**, the iterations are divided by the number of threads. In this case, there will be 4 blocks of the same size (it may occur that the last block does not have the same size). The first block will correspond to the first thread, and successively. Thread 0 will execute block 0 (0-2 iterations), thread 1 block 1 (3-5 iterations), thread 2 will execute block 2 (6-8 iterations) and thread 3 will execute block 4 (9-11 iterations).

We can enforce to create blocks of 2 iterations with **schedule(static, 2)**. Thread 0 will execute blocks 0 (0-1 iterations) and 4 (8-9 iterations). Thread 1 will execute 1 (2-3 iterations) and 5 (10-11 iterations). Thread 2, block 2 (4-5 iterations) and thread 3, block 3 (6-7 iterations).

The **schedule(dynamic, 2)** divides the iterations in blocks of 2 iterations, but it distributes the blocks in a different way than static. It gives the next block to execute to the first thread to end its work. This is a better solution, because it is possible that some thread has a lot of work in a block and has to execute another one. With dynamic, the other threads will execute that block. We cannot say which threads will execute each block, but we can say that for each block, the thread will execute two iterations of the loop.

The **schedule(guided, 2)** gives n/p iterations to the first loop, (n - n/p)/p iterations to the next loop and so on until blocks have 2 iterations. Then, it starts to work as a dynamic schedule.

2.2.2 2.nowait.c

1. Which could be a possible sequence of printf when executing the program?

In this code, two threads will execute the first loop while the other two threads execute the second loop. A possible sequence of prints would be the following:

```
Loop 1: thread (0) gets iteration 0
Loop 1: thread (3) gets iteration 1
Loop 2: thread (1) gets iteration 2
Loop 2: thread (2) gets iteration 3
```

2. How does the sequence of printf change if the nowait clause is removed from the first for directive?

Now, while two threads are executing the first loop, the other two will remain doing nothing. Once they end with the loop, only 2 threads will enter in the second loop. As the schedule type is dynamic, we do not know which two threads will execute the for loops.

3. What would happen if dynamic is changed to static in the schedule in both loops? (keeping the nowait clause)

If we change it to static, threads 0 and 1 will execute the first and the second for loop because threads 2 and 3 will not receieve any work.

2.2.3 3.collapse.c

1. Which iterations of the loop are executed by each thread when the collapse clause is used?

The **collapse** clause specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. In this code, there is no schedule clause, so it will be treated as static.

```
Thread 0: 00, 01, 02, 03
Thread 1: 04, 10, 11
Thread 2: 12, 13, 14
Thread 3: 20, 21, 22
Thread 4: 23, 24, 30
Thread 5: 31, 32, 33
Thread 6: 34, 40, 41
Thread 7: 42, 43, 44
```

Where 00 means i = 0 and j = 0, 01 means i = 0 and j = 1...

2. Is the execution correct if the collapse clause is removed? Which clause (different than collapse) should be added to make it correct?

The execution is not correct because some iterations are not executed. The main reason is because there are data racing conflicts with variable *j*. To avoid this, we should make private the *j* variable with #pragma omp parallel for private(j).

2.3 Synchronization

2.3.1 1.datarace.c

(execute several times before answering the questions)

1. Is the program always executing correctly?

2. Add two alternative directives to make it correct. Explain why they make the execution correct.

2.3.2 2.barrier.c

1. Can you predict the sequence of messages in this program? Do threads exit from the barrier in any specific order?

2.3.3 3.ordered.c

1. Can you explain the order in which the "Outside" and "Inside" messages are printed?

2. How can you ensure that a thread always executes two consecutive iterations in order during the execution of the ordered part of the loop body?

2.4 Tasks

2.4.1 1.single.c

1. Can you explain why all threads contribute to the execution of instances of the single worksharing construct? Why are those instances appear to be executed in bursts?

Thanks to the **single** clause only 1 thread executes each iteration of the loop, and thanks to the **nowait** clause, the other threads do not have to wait until the iteration ends to continue with the other iterations.

The 4 threads distribute the first 4 iterations. They print a message and remain sleeping during 1 second. Then, they distribute again the 4 next iterations, and so on. As a consequence, we see the messages in bursts.

2.4.2 2.fibtasks.c

1. Why all tasks are created and executed by the same thread? In other words, why the program is not executing in parallel?

The program has 0 parallel regions, so only 1 thread is executing the code. Thus, only 1 thread creates tasks and executes them.

2. Modify the code so that the program correctly executes in parallel, returning the same answer that the sequential execution would return.

FALTA PARLAAAAAAAAAAAAAAAAAAR DEL PER QUE HEM FET AIXO!?!?!?!?!?

```
#pragma omp parallel
#pragma omp single
while (p != NULL) {
      printf("Thread %d creating task that will compute %d\n",
             omp_get_thread_num(), p->data);
      #pragma omp task firstprivate(p)
      processwork(p);
      p = p->next;
}
```

2.4.3 3.synchtasks.c

1. Draw the task dependence graph that is specified in this program

The **depend** clause establish dependences, only between tasks, on the scheduling of tasks. There are different types of dependence:

- **in**: The task has to wait until all previously generated sibling tasks that reference at least one of the list items in an *out* or *inout* list end their execution.
- **out** and **inout**: The task has to wait untill all previously generated sibling tasks that reference at least one of the list items in an *in*, *out* or *inout* list end their execution.

In the given code, the first task has an **out** depende type, with only the variable a in its list. As there are no previously generated sibling tasks, this first task is not dependent of any task. Tasks 2 and 3 are not dependent of any previously generated sibling task. Although there exist some previously sibling tasks, none of them has variables b or c in its lists.

On the other hand, task 4 has an **in** dependence type with variables a and b and an **out** dependence type with variable d. Thus, this task will have to wait until tasks 1 and 2 end their execution.

Finally, task 5 has an **in** dependence type with variables c and d. This taks will depend from tasks 3 and 4. Hence, the task dependence graph is the following:

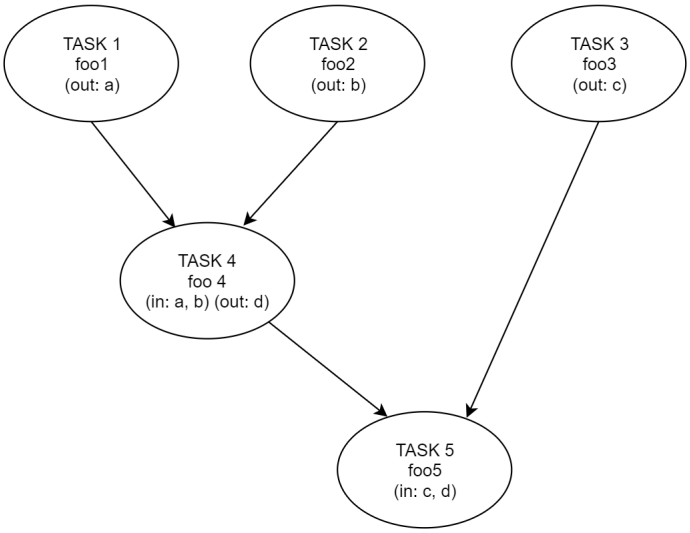

Figure 1: Task dependency graph of code 3.synchtasks.c.

2. Rewrite the program using only taskwait as task synchronisation mechanism (no depend clauses allowed)

The **taskwait** construct makes a task wait until the completion of its child tasks. The new version of the code is shown below. Task 4 will wait until tasks 1 and 2 end their execution and task 5 will wait until the completition of tasks 3 and 4. Doing this we recreated the task dependency graph of Figure 1 without using the **depend** clause.

The first problem we have seen is that task 3 cannot be executed at the same time than tasks 1 and 2. Otherwise, task 4 would have to wait until tasks 1, 2 and 3 end their execution altough it does not depend from task 3. The second problem is that with this code, we are not able to create the 5 tasks at the same time. Tasks 1, 2 and 4 will be created first. Then tasks 3 and 5.

```
    printf("Creating task foo1\n");
    #pragma omp task
    foo1();

    printf("Creating task foo2\n");
    #pragma omp task
    foo2();

    printf("Creating task foo4\n");
    #pragma omp taskwait
    foo4();

    printf("Creating task foo3\n");
    #pragma omp task
    foo3();

    printf("Creating task foo5\n");
    #pragma omp taskwait
    foo5();
```

Figure 2: Modified version of the given code.

Figures 3 and 4 show the output when executing the two versions of the code. We can see clearly the differences between one and the other explained before.

```
Creating task foo1
Creating task foo2
Creating task foo3
Creating task foo4
Creating task foo5
Starting function foo3
Starting function foo1
Starting function foo2
Terminating function foo1
Terminating function foo2
Starting function foo4
Terminating function foo4
Terminating function foo3
Starting function foo5
Terminating function foo5
```

Figure 3: Output of the original version of the code.

```
Creating task foo1
Creating task foo2
Creating task foo4
Starting function foo2
Starting function foo1
Terminating function foo2
Terminating function foo1
Starting function foo4
Terminating function foo4
Creating task foo3
Creating task foo5
Starting function foo3
Terminating function foo3
Starting function foo5
Terminating function foo5
```

Figure 4: Output of the modified version of the code.

2.4.4 4.taskloop.c

1. Find out how many tasks and how many iterations each task execute when using the grainsize and num tasks clause in a taskloop. You will probably have to execute the program several times in order to have a clear answer to this question.

The **grainsize(grain-size)** clause controls how many loop iterations are assigned to each created task. The number of loop iterations assigned to each created task is greater than or equal to the minimum of the value of grain-size and the total number of iterations.

We though that when using the **grainsize(5)** clause there will be created 3 tasks, the first two executing 5 iterations and the other one 2. In reality, there are only 2 tasks created, each one executing 6 iterations. This is because the number 5 does not mean that each task has 5 iterations but a minimum of 5, unless there remain less than 5 iterations.

The **num_tasks(num-tasks)** clause creates as many tasks as the minimum of num-tasks and the number of loop iterations.

There are 5 tasks created. The first two execute 3 iterations of the loop whereas the other 3 execute only 2.

```
Going to distribute 12 iterations with grainsize(5) ...
Loop 1: (1) gets iteration 0
Loop 1: (1) gets iteration 1
Loop 1: (1) gets iteration 2
Loop 1: (1) gets iteration 3
Loop 1: (2) gets iteration 6
Loop 1: (2) gets iteration 7
Loop 1: (2) gets iteration 8
Loop 1: (2) gets iteration 9
Loop 1: (2) gets iteration 10
Loop 1: (2) gets iteration 11
Loop 1: (1) gets iteration 4
Loop 1: (1) gets iteration 5
Going to distribute 12 iterations with num_tasks(5) ...
Loop 2: (2) gets iteration 0
Loop 2: (2) gets iteration 1
Loop 2: (1) gets iteration 6
Loop 2: (1) gets iteration 7
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
Loop 2: (1) gets iteration 8
Loop 2: (1) gets iteration 9
Loop 2: (2) gets iteration 2
Loop 2: (3) gets iteration 3
Loop 2: (3) gets iteration 4
Loop 2: (3) gets iteration 5
```

Figure 5: Ouput of the code without the nogroup clause.

2. What does occur if the nogroup clause in the first taskloop is uncommented?

The **taskloop** clause implicitly generates a **taskgroup** region that encloses it. In this region there is a barrier at the end, so the second loop will not be executed until the first one terminates. The **nogroup** clause removes the **taskgroup** region.

With the **nogroup** clause, the second loop can be executed at the same time than the first loop.

Now, there are created the same number of tasks, and each one does the same itreations than in the previous question. However, while two threads execute the first loop, the other two threads will execute the 5 tasks of the second loop (Figure 6), unless one of the two threads of the first loop end before the second loop terminates. In this situation, this thread will also execute tasks of the second loop (Figure 7).

```
Going to distribute 12 iterations with grainsize(5) ...
Going to distribute 12 iterations with num_tasks(5) ...
Loop 1: (1) gets iteration 0
Loop 1: (1) gets iteration 1
Loop 1: (2) gets iteration 6
Loop 1: (2) gets iteration 7
Loop 1: (2) gets iteration 8
Loop 1: (1) gets iteration 2
Loop 2: (0) gets iteration 10
Loop 2: (0) gets iteration 11
Loop 2: (0) gets iteration 8
Loop 2: (3) gets iteration 0
Loop 2: (3) gets iteration 1
Loop 2: (3) gets iteration 2
Loop 2: (3) gets iteration 3
Loop 2: (3) gets iteration 4
Loop 2: (3) gets iteration 5
Loop 2: (3) gets iteration 6
Loop 2: (3) gets iteration 7
Loop 1: (1) gets iteration 3
Loop 1: (1) gets iteration 4
Loop 1: (1) gets iteration 5
Loop 2: (0) gets iteration 9
Loop 1: (2) gets iteration 9
Loop 1: (2) gets iteration 10
Loop 1: (2) gets iteration 11
```

Figure 6: Ouput 1 of the code with the nogroup clause.

```
Going to distribute 12 iterations with grainsize(5) ...
Going to distribute 12 iterations with num_tasks(5) ...
Loop 1: (2) gets iteration 0
Loop 1: (2) gets iteration 1
Loop 1: (2) gets iteration 2
Loop 1: (2) gets iteration 3
Loop 1: (2) gets iteration 4
Loop 1: (2) gets iteration 5
Loop 1: (1) gets iteration 6
Loop 1: (1) gets iteration 7
Loop 1: (1) gets iteration 8
Loop 2: (0) gets iteration 10
Loop 2: (3) gets iteration 3
Loop 1: (1) gets iteration 9
Loop 1: (1) gets iteration 10
Loop 1: (1) gets iteration 11
Loop 2: (0) gets iteration 11
Loop 2: (0) gets iteration 8
Loop 2: (0) gets iteration 9
Loop 2: (3) gets iteration 4
Loop 2: (3) gets iteration 5
Loop 2: (1) gets iteration 6
Loop 2: (1) gets iteration 7
Loop 2: (2) gets iteration 0
Loop 2: (2) gets iteration 1
Loop 2: (2) gets iteration 2
```

Figure 7: Ouput 2 of the code with the nogroup clause.

3 Observing overheads

3.1 Synchronisation overheads

In this part we have different ways of computing the pi number, each with a different method of synchronization of the sum variable.

3.1.1 pi_sequential

```
Wall clock execution time  = 1.793329954 seconds
Value of pi = 3.1415926536
```

Figure 8: Output of pi_sequential

3.1.2 pi_omp_critical

```
#pragma omp parallel private(x) num_threads(num_threads)
{
    #pragma omp for
    for (long int i=0; i<num_steps; ++i) {
        x = (i+0.5)*step;
        #pragma omp critical
        sum += 4.0/(1.0+x*x);
    }
}
```

Figure 9: Parallel code (pi_omp_critical)

3.1.2.1 1 thread

```
Total execution time: 4.319360s
Number pi after 100000000 iterations = 3.141592653590426
```

Figure 10: Output of pi_omp_critical with 1 thread

3.1.2.2 4 threads

```
Total execution time: 38.100659s
Number pi after 100000000 iterations = 3.141592653590203
```

Figure 11: Output of pi_omp_critical with 4 threads

3.1.2.3 8 threads

```
Total execution time: 34.410379s
Number pi after 100000000 iterations = 3.141592653589771
```

Figure 12: Output of pi_omp_critical with 8 threads

3.1.3 pi_omp_atomic

```
#pragma omp parallel private(x) num_threads(num_threads)
{
    #pragma omp for
    for (long int i=0; i<num_steps; ++i) {
        x = (i+0.5)*step;
        #pragma omp atomic
        sum += 4.0/(1.0+x*x);
    }
}
```

Figure 13: Parallel code (pi_omp_atomic)

3.1.3.1 1 thread

```
Total execution time: 1.794664s
Number pi after 100000000 iterations = 3.141592653590426
```

Figure 14: Output of pi_omp_atomic with 1 thread

3.1.3.2 4 threads

```
Total execution time: 6.201648s
Number pi after 100000000 iterations = 3.141592653590225
```

Figure 15: Output of pi_omp_atomic with 4 threads

3.1.3.3 8 threads

```
Total execution time: 6.910445s
Number pi after 100000000 iterations = 3.141592653589742
```

Figure 16: Output of pi_omp_atomic with 8 threads

3.1.4 pi_omp_reduction

```
#pragma omp parallel private(x) num_threads(num_threads) reduction(+:sum)
{
    #pragma omp for
    for (long int i=0; i<num_steps; ++i) {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
}
```

Figure 17: Parallel code (pi_omp_reduction)

3.1.4.1 1 thread

```
Total execution time: 1.831957s
Number pi after 100000000 iterations = 3.141592653590426
```

Figure 18: Output of pi_omp_reduction with 1 thread

3.1.4.2 4 threads

```
Total execution time: 0.478957s
Number pi after 100000000 iterations = 3.141592653589683
```

Figure 19: Output of pi_omp_reduction with 4 threads

3.1.4.3 8 threads

```
Total execution time: 0.261439s
Number pi after 100000000 iterations = 3.141592653589815
```

Figure 20: Output of pi_omp_reduction with 8 threads

3.1.5 pi_omp_sumlocal

```
double sumlocal = 0.0;
#pragma omp parallel private(x) firstprivate(sumlocal) num_threads(num_threads)
{
    #pragma omp for
    for (long int i=0; i<num_steps; ++i) {
        x = (i+0.5)*step;
        sumlocal += 4.0/(1.0+x*x);
    }
    #pragma omp critical
    sum += sumlocal;
}
```

Figure 21: Parallel code (pi_omp_sumlocal)

3.1.5.1 1 thread

```
Total execution time: 1.832351s
Number pi after 100000000 iterations = 3.141592653590426
```

Figure 22: Output of pi_omp_sumlocal with 1 thread

3.1.5.2 4 threads

```
Total execution time: 0.482865s
Number pi after 100000000 iterations = 3.141592653589683
```

Figure 23: Output of pi_omp_sumlocal with 4 threads

3.1.5.3 8 threads

```
Total execution time: 0.261369s
Number pi after 100000000 iterations = 3.141592653589815
```

Figure 24: Output of pi_omp_sumlocal with 8 threads

3.2 Thread creation and termination

Code *pi_omp_parallel.c* is used to calculate the thread creation and termination overhead. To do this, the code has 2 parts which computes the same value (pi), one sequential and the other parallel. The tricky part is that in the parallel region only 1 thread executes the code, so it takes the same time than in the sequential part.

As a result, the difference between the sequential and the parallel regions will only be caused by the overheads.

The order of magnitud of the overhead generated in the threads creation is in microseconds. We can see it clearly in Figure 25 because the first print is a message with the order of magnitude.

```
All overheads expressed in microseconds
Nthr    Overhead        Overhead per thread
2       2.0023          1.0011
3       1.6780          0.5593
4       1.7134          0.4284
5       1.7847          0.3569
6       1.8046          0.3008
7       1.7323          0.2475
8       1.7549          0.2194
9       2.2797          0.2533
10      2.3004          0.2300
11      2.3724          0.2157
12      2.4019          0.2002
13      2.4509          0.1885
14      2.5782          0.1842
15      2.7435          0.1829
16      2.7500          0.1719
17      2.8480          0.1675
18      2.8822          0.1601
19      2.8254          0.1487
20      2.9179          0.1459
21      2.9622          0.1411
22      3.0387          0.1381
23      3.0811          0.1340
24      3.7695          0.1571
```

Figure 25: Output of pi_omp_parallel.c (execution in queue)

On the one hand, we see that the bigger the number of threads used, the bigger the overhead. This is because we have to create and terminate more threads.

On the other hand, the overhead per thread decreases as we increase the number of threads. **REPASSSSSSSSSSSSSSSSSSSSSAAAAAAAAAAAAAAAAAAAAAAAA** We think that this may happen because the task of computing the pi number if bigger than the creation and termination overheads, so adding more threads will still take less time than the pi task. Consequently, the total time in the parallel region will remain constant and this number divided by a bigger number gives a smaller number. We think that if we execute the code with a sufficient number of threads, in the end, the thread overheads would pass the task time and the overhead per thread would begun to increase a little.

3.3 Task creation and synchronization

In this section, we studied the overhead when creating tasks and synchronizing them.

The order of magnitude is also in microsecond. It is shown in the first line of Figure 26. The execution was using 10 tasks and 1 thread. Doing this, we only have the overhead of the creation (**task)** and synchronization (**taskwait**) of the tasks if we compute $T_{par} - T_{seq}$.

```
All overheads expressed in microseconds
Ntasks  Overhead        Overhead per task
2       0.2318          0.1159
4       0.5196          0.1299
6       0.7690          0.1282
8       1.0167          0.1271
10      1.2827          0.1283
12      1.5125          0.1260
14      1.7640          0.1260
16      2.0038          0.1252
18      2.2672          0.1260
20      2.5160          0.1258
22      2.7558          0.1253
24      3.0027          0.1251
26      3.2645          0.1256
28      3.4966          0.1249
30      3.7534          0.1251
32      3.9903          0.1247
34      4.2385          0.1247
36      4.4821          0.1245
38      4.7268          0.1244
40      4.9663          0.1242
42      5.2138          0.1241
44      5.4777          0.1245
46      5.7102          0.1241
48      5.9602          0.1242
50      6.2070          0.1241
52      6.4543          0.1241
54      6.7144          0.1243
56      6.9368          0.1239
58      7.2039          0.1242
60      7.4387          0.1240
62      7.7059          0.1243
64      7.9344          0.1240
```

Figure 26: Execution of pi_omp_tasks.c (queue execution).

The total overhead increases when there are more tasks because we have to create and synchronize more tasks. However, the overhead per tasks is constant due to the fact that each task has the same execution time. As we are only using 1 thread, the tasks cannot be executed in parallel, so adding the time/task and dividing it by 1 more task leaves the result constant.

$Overhead\_per\_task \approx 0.125$

$Overhead = \#tasks * overhead/task$

$Overhead\_per\_task = Overhead/\#tasks \approx 0.125$